TNE80013 Software Managed Networks Final Project - Portfolio Task

Leo Holden

Dept. Of Computing Technologies

Swinburne University of Technology

Hawthorn, Australia

103996982@student.swin.edu.au

Due Date: 29-10-2025

Date of Submission: 29-10-2025

Abstract—This project is written to demonstrate a proof-of-concept expansion in SHS2 branch networks using SDN principles. The configurations of devices are documented in this report along with the tools and techniques used to achieve successful implementation of requirements. There are high level requirements and constraints set by the client SHS2 and are directly met with appropriate design and implementation choices to best adhere to the SHS2 requirements. The deployment takes into account the current network deployment and organizations usage and the scalability to ensure future infrastructure can be implemented for enhancements in the system. Furthermore, additional services are proposed and configured in the proof -of-concept deployment to reflect these possibilities for future developments. These include overlay tunnelling technologies, packet manipulation by means of manually installed flows essentially acting as a NAT device, firewall packet filtering using manually installed flows providing branch security, port mirroring and traffic analysis using NTOPNG and automation of the prototype topology deployment using a Python script.

I. INTRODUCTION

A. Background

Stay Home Stay Safe (SHS2) is an Australian video production company with three isolated branches with an expanding number of employees in each. The main requirement connecting the three branches using through a private network supporting a fast as possible network with high-bandwidth video streaming.

B. Objective

To deploy a proof-of-concept (POC) network design, it is implemented in GNS3. The solution is to be Software Defined Networking (SDN) based, to satisfy the following requirements:

- *End to end connectivity to and from every branch*
- *Flexible and scalable network design*
- *Traffic filtering with firewalls*
- *Network monitoring systems*
- *Automated control process of operation*

C. Tools and Technologies

A virtualized environment is created in VMware by launching a Linux virtual machine (VM). Preinstalled with the tools, running a 64-bit Ubuntu version 18.04.5 LTS. The VM has been allocated the following hardware resources: 8 GB ram, 8 cores and a 40 GB disk partition. Preinstalled is GNS3 version 2.2.22 with the Open vSwitch (OVS) support OpenFlow version 1.0-1.5. *Table 1* shows the resource specifications of the local machine and VM.

Specification	**Local Machine**	**Virtual Machine**
CPU	12th Gen Intel i5-12500H (12 cores / 16 threads)	8 virtual CPUs allocated
RAM	16 GB	8 GB
Storage	512 GB	40 GB
Operating System	Windows 11 64-bit	Ubuntu 18.04.5 LTS 64-bit
Virtualization Tool	VMware Workstation 17	
Software Versions		GNS3 2.2.22, Open vSwitch 2.17.12, OpenFlow 1.0-1.5, Python 2.7.17

Table 1. Local hardware and virtual machine specifications

The SDN network will use both Ryu [3] and ONOS [4] controllers to establish the control planes, Iperf3 will be utilised to conduct testing on varying network conditions and parameters and Python to develop automations quickly. Additionally, network firewalls are introduced to address security concerns and provide filtering on traffic when entering other branches. NTOPNG will be utilised to monitor the network statistics and health.

D. Report Overview

The following section discusses the methodology and overarching approach in the system's architecture and why to use SDN. Section III sees the implementation of the requirements, each subsection discusses and shows the deployment plan or changes, specific configurations and commands, specific file modifications, screenshots of results and any issues had in deployment and the resolution procedure.

II. METHODOLOGY OVERVIEW

A. SDN based networking solution

Traditional networking devices have three planes: management, control and data plane. SDN decouples the control plane (decision processing) and data plane (packet forwarding). The control plane is centralised in a controller that governs the routing of packets by installing flows with OpenFlow protocol. This is done programmatically and does not require physical access to the device. A controller itself is configurable programmatically; a northbound API allows communication to control applications that influence the network. A southbound API is used to communicate these changes to the networking devices. This is applicable in this report where a Ryu and ONOS controller are used.

The basis of the network architecture is founded on the underlay network which is the physical network connections. Connecting branches becomes the task of building logical tunnels from each branch that traverses the underlying underlay network, enabled by tunnelling protocols Generic Routing Encapsulation (GRE) and Virtual Extensible Local Area Network (VXLAN). This approach enables extending the layer 2 connectivity across layer 3 infrastructure.

B. GNS3 and Network Appliances

To implement the prototype network design, an emulator GNS3 [2] is used to provide an environment where appliances emulate real Operating System (OS) behaviour. GNS3 has a large community backing it with predefined network appliances out the box ready, available to download and utilise without OS installation troubles.

III. IMPLEMENTATION AND RESULTS

This section documents the network design and its configurations completed in GNS3 with supporting documentation material referenced for further information. Further background information is given about SHS2's existing infrastructure and network requirements. Information on importing appliances in GNS3 can be found in [2].

Figure 1 shows the existing network topology and the respective interfaces and IP configurations, amongst the other implementations. A VXLAN tunnel connects Branch1 and Branch2, Branch1 and Branch3 are connected by GRE tunnel. The ISP router is connected to a Ryu controller whereas the SHS2 OVS switches are managed by a ONOS controller. `Mgmt-Switches` are traditional switches that are the connection point for the controller running locally on the host machine

2) Configuration details

IP configurations are required to be static and persist restarts, this is achieved by editing the file located `/etc/networks/interfaces`. *Figure 2* demonstrates which lines to uncomment; this can be done for each device

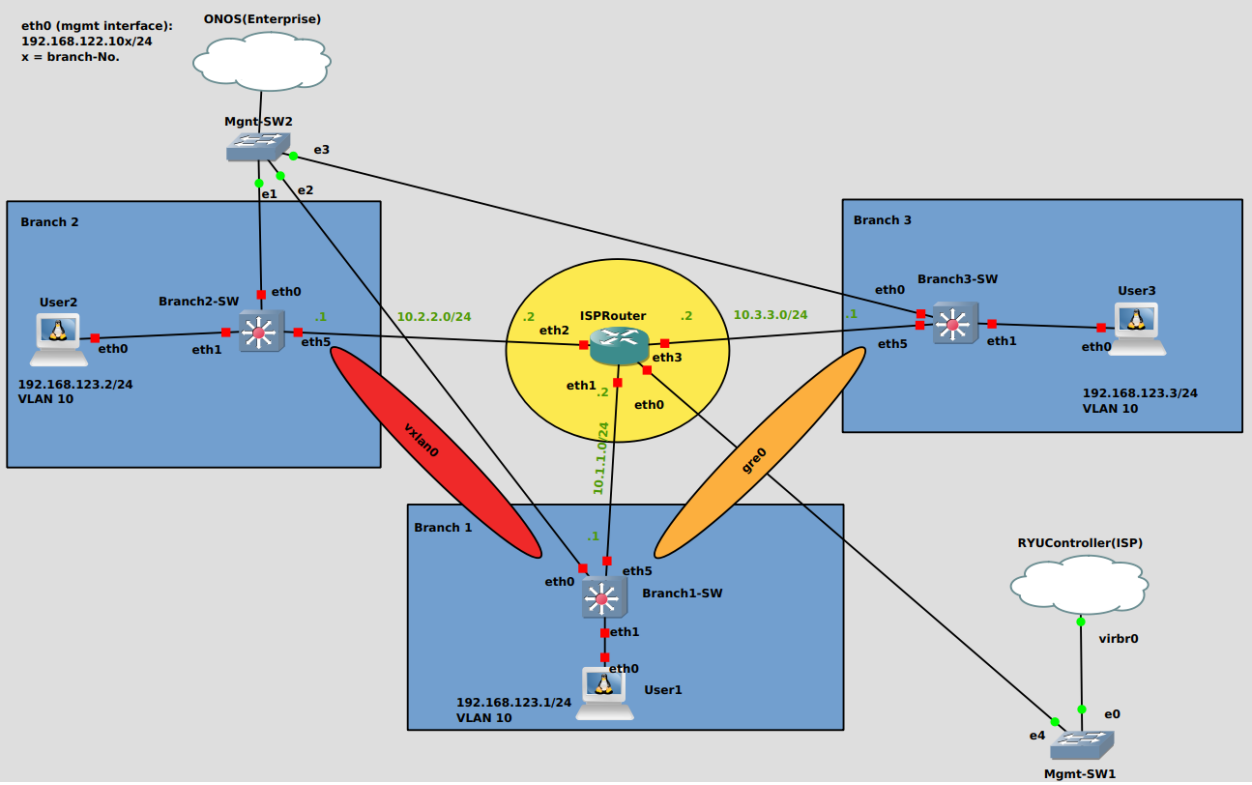

Figure 1. Existing Networking Connectivity Topology

A. Existing Network Connectivity

1) Deployment Plan

SHS2 has three branches currently with an occupying max number of users per branch at one. Three branches operate in the `192.168.123.0/24` subnet. IP configurations, where *x* indicates the respective branch identifier value for each link:

- `Branch1 (user1) 192.168.123.1/24`
- `Branch2 (user2) 192.168.123.2/24`
- `Branch3 (user3) 192.168.123.2/24`
- `Branch1 (link to ISP): 10.1.1.1/24`
- `Branch2 (link to ISP): 10.2.2.1/24`
- `Branch3 (link to ISP): 10.3.3.1/24`
- `ISP (links to branches): 10.x.x.2/24`

except ISP router interfaces. The appliance requires restarting to apply the changes and configuration can be verified by issuing the Command Line interface (CLI) command: `ifconfig [interface]`.

```
# This is a sample network config uncomment lines to configure the network
#
# Static config for eth0
auto eth0
iface eth0 inet static
        address 192.168.123.2
        netmask 255.255.255.0
#       gateway 10.2.2.2
#       up echo nameserver 192.168.0.1 > /etc/resolv.conf
```

Figure 2. User2 interfaces file lines to uncomment

The cloud object appliance connects the local machine to become the controller for the network. The `virbr0` interface

is automatically given the IP address of `192.168.122.1/24`. The special interface is applied by ticking the show special interfaces and then selecting it from the drop-down selection and applying the changes. The management interface of each OVS (eth0) is as follows:

- `Branch1-SW: 192.168.122.101/24`
- `Branch2-SW: 192.168.122.102/24`
- `Branch3-SW: 192.168.122.103/24`

The default-gateway of each branch switch should also point to the ISP router and can be configured as per *Figure 2:*

- `Branch1-SW default-gateway: 10.1.1.2`
- `Branch2-SW default-gateway: 10.2.2.2`
- `Branch3-SW default-gateway: 10.3.3.2`

In this deployment, `br0` is used as the networking bridge that across OVS switches, the following commands can be used to delete and add a bridge: `ovs-vsctl del-br | add-br [bridge]`, in this instance all bridges are deleted and then `br0` is added. To verify configurations the CLI command: `ovs-vsctl show`, See *Figure 3.*

```
Branch2-SW:/$ ovs-vsctl show
7830aa27-938c-4244-a16c-a9e53e4db085
    Bridge br0
        Controller "tcp:192.168.122.1:7777"
        Port eth1
            tag: 10
            Interface eth1
        Port br0
            Interface br0
                type: internal
```

Figure 3. ovs-vsctl show command output (Branch2 OVS)

To add interfaces to be part of a bridge, use the CLI command: `ovs-vsctl add-port [bridge] [port] tag=[vlan-id]`. 802.1q VLAN tagging is enabled by tagging the port, to add a port without tagging the parameter is omitted. Additionally, to prevent CPU spikes in GNS3 resulting from broadcast loops, issuing the command: `ovs-vsctl set bridge br0 stp_enable=true`. The following list is the complete commands issued regarding `br0`:

- `ovs-vsctl del-br [bridge]`
- `ovs-vsctl add-br [bridge]`
- `ovs-vsctl add-port [bridge] [port] tag=[vlan-id]`
- `ovs-vsctl add-port [bridge] [port]`

Port `eth0` is not included in the bridge as best practise because it acts as the management connection. The interfaces that are required to be added to `br0` for each OVS switch:

- `Foreach branch OVS switch: eth1`
- `ISP Router: eth1-3`

OpenFlow version 1.5 is not supported by the controllers used in this project within GNS3. Therefore, forcing OpenFlow versions on the bridge to prevent connection errors: `ovs-vsctl set bridge br0 protocols=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13,OpenFlow14`. The Ryu and ONOS controller are ran on the local machine hosting GNS3, by default both applications operate on port 6653 for OpenFlow protocol. The running port number for a ONOS controller is modified and as per SHS2 requirement should be set to port 7777. Before modification, the OVS switches can have the controller set, see *Figure 3* for verification. Furthermore, the second command below also provides a way to verify switch to controller connectivity.

- `ovs-vsctl set-controller [bridge] tcp:[ip-address]:[port-number]`
- `ovs-vsctl get-controller [bridge]`

The IP address for all switches is configured to local VM's hostname, however, as per the topology in *Figure 1,* Ryu controller is set for the ISP router and ONOS for the SHS2 branches. Therefore, port number 7777, is set for the SHS2 branch switches. To run the Ryu and ONOS controllers, the following commands start the controllers:

- `sudo /opt/onos/bin/onos-service start`
- `Ryu-manager --verbose rest_router.py gui_topology/gui_topology.py`

(Note: ensure the Ryu command for startup is issued in the correct directory where the Ryu-manager is located). To modify the running port number for the ONOS controller, the application can be connected via SSH through with the following commands and verified:

- `ssh -p 8101 -o StrictHostKeyChecking=no onos@localhost`
- `Login password; rocks`
- `Cfg set org.onosproject.openflow .controller.impli.OpenFlowControllerImpl openflowPorts [port-number]`
- `Sudo lsof -i -P -n | grep LISTEN`

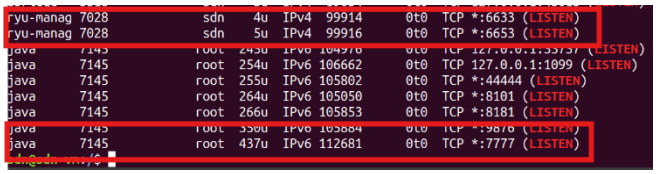

Figure 4. ONOS and Ryu controller listening on desired ports

The `rest_router.py` parameter in the Ryu controller start up command exposes a northbound REST API for control plane management. The data plane interfaces can be configured through commands initiated on the local host:

- `curl -X POST -d '{"address":"10.1.1.2/24"}' http://localhost:8080/router/[SW-id]`
- `curl -X POST -d '{"address":"10.2.2.2/24"}' http://localhost:8080/router/[SW-id]`

- `curl -X POST -d '{"address":"10.3.3.2/24"}' http://localhost:8080/router/[SW-id]`

To discover the *SW-id*, from the terminal of which Ryu controller was started from, the Ryu controller terminal output or the command: `curl -X GET http://localhost:8080/stats/switches`. The output is a large integer that is required to be converted to a 16-digit hexadecimal to be used as the switch identifier value.

Figure 5. OVS switch-id in the Ryu manager output

To enable OpenFlow on the ONOS controller the configuration can be made in the ONOS GUI, more information regarding ONOS connected OVS information can be found here. Similarly, the Ryu controller GUI provides a topology visualization and flows tables for each node.

- `http://localhost:8181/onos/ui`
- `http://localhost:8080`

To enable OpenFlow protocol on the ONOS controller activate the following applications in the ONOS GUI: `OpenFlow Provider Suite, Reactive Forwarding.`

GRE and VXLAN tunnel creation is configured on each endpoint of the tunnel on the OVS switches, the `ovs-vsctl show` command can be used to verify tunnel configuration.

- `ovs-vsctl add-port br0 vxlan0 tag=10 -- set interface vxlan0 type=vxlan options:key=10 options:remote_ip=[endpoint-ip]`
- `ovs-vsctl add-port gre0 -- set interface gre0 type=gre options:remote_ip=[endpoint-ip]`

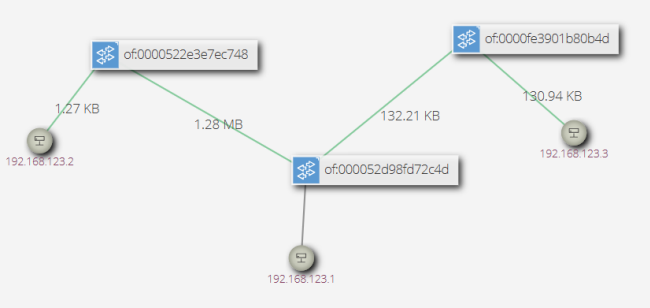

Figure 6. ONOS GUI topology

Figure 6 shows the SHS2 overlay network, the topology illustrates the topology that the Branch1 is directly connected. However, the connection traverses across the ISP router and this topology shows the logical link created by the tunnelling protocols. The flows can be seen on each switch through the command: `ovs-ofctl dump-flows br0` or can be viewed in each controller in the GUI. (Note: Initiating pings from each host will make the host appear on the ONOS GUI topology given you have selected the option to show hosts)

3) Testing

Pings from each host are successfully reaching each other hosts, viewing the flow tables for both controllers indicate that newly installed flows packet count field is increasing. Furthermore, link usage can be toggled and seen when generating traffic, see *Figure 6*. Generating traffic using Iperf3 [5] is unsuccessful using default parameters, to successfully generate traffic using Iperf3, set the Maximum Segment Size (MSS) to a value less than 1410.

- `Iperf3 -s`
- `Iperf3 -c [server-ip] -M [MSS-value]`

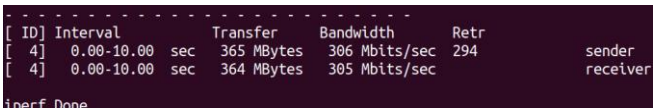

Figure 7. Iperf3 results (MSS set to 1410)

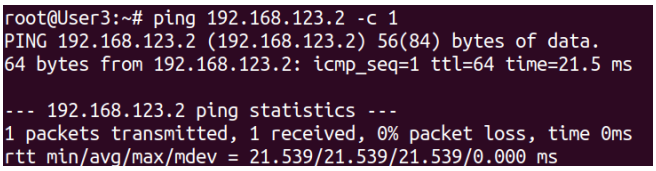

Figure 8. Ping results (user3 to user2)

4) Discussion

The Iperf3 testing failing was due to the ethernet Maximum Transmission Unit (MTU) being 1500 bytes, this can be verified through `ifconfig` command. VXLAN adds 50 additional bytes in the encapsulation header [6] and GRE includes an additional 24 bytes in the encapsulation header [7]. This results in testing with Iperf3 to complete the TCP connection, however no data is sent across after the TCP handshake. 40 bytes is assigned to the IP and TCP headers of packets; therefore, an MSS typically is 1460, however with the tunnelling headers included this pushes up the limit [5]. Therefore, deducing the 1410 value results from the additional 50 bytes. This can be verified by setting the MTU values on the ethernet interfaces and tunnels to 1600, to ensure configuration remains persistent ethernet interfaces MTU values are set in the file found at `/etc/networks/interfaces` and inserting `mtu 1600`.

- `ovs-vsctl set Interface [vxlan0 | gre0] mtu_request=1600`
- `ovs-vsctl list interface`
- `ifconfig [interface] | grep MTU`

ISP router IP configurations are completed through the Ryu rest router API and is not yet automated to become persistent. Therefore, the following commands is required to set MTU value for eth1, eth2, eth3 on ISP router. (Note: it may be more

realistic to limit the branch switches MTU because we would not have controller over ISP router)

- `ip link set dev [interface] mtu 1600`

```
# Static config for eth5
auto eth5
iface eth5 inet static
        address 10.2.2.1
        netmask 255.255.255.0
        gateway 10.2.2.2
        mtu 1600
```

Figure 9. Persistent MTU configuration

B. *New Servers*

1) *Design plan*

Branches 2 and 3 are to receive a new server and require direct communication, the servers are to be in VLAN 20, with the expectation that the servers will be streaming video traffic. To accomplish this task, a VXLAN tunnel is created across branches 2 and 3. Round Trip Time (RTT) must be minimised; GRE tunnelling has less overhead (24 bytes) relative to VXLAN (50 bytes). GRE is an older protocol compared to VXLAN which is purposed for SDN overlay networks.

- 1476 / 1500 x 100 = 98.4%
- 1450 / 1500 x 100 = 96.7%
- VXLAN is 1.7% less efficient

2) *Configuration Details*

The connecting ports foreach server to its respective branch switch is `eth2` and therefore are required to be tagged as VLAN 20 and added to the `br0`. The `vxlan1` tunnel links Branches 2 and 3, the tunnel interface should have its MTU size adjusted:

- `ovs-vsctl add-port br0 eth2 tag=20`
- `ovs-vsctl add-port my-br vxlan1 tag=20 -- set interface vxlan1 type=vxlan options:key=20 options:remote_ip=10.2.2.1`
- `ovs-vsctl add-port my-br vxlan1 tag=20 -- set interface vxlan1 type=vxlan options:key=20 options:remote_ip=10.3.3.1`
- `ovs-vsctl set Interface vxlan1 mtu_request=1600`
- `Server2: 172.16.123.2/24`
- `Server3: 172.16.123.3/24`

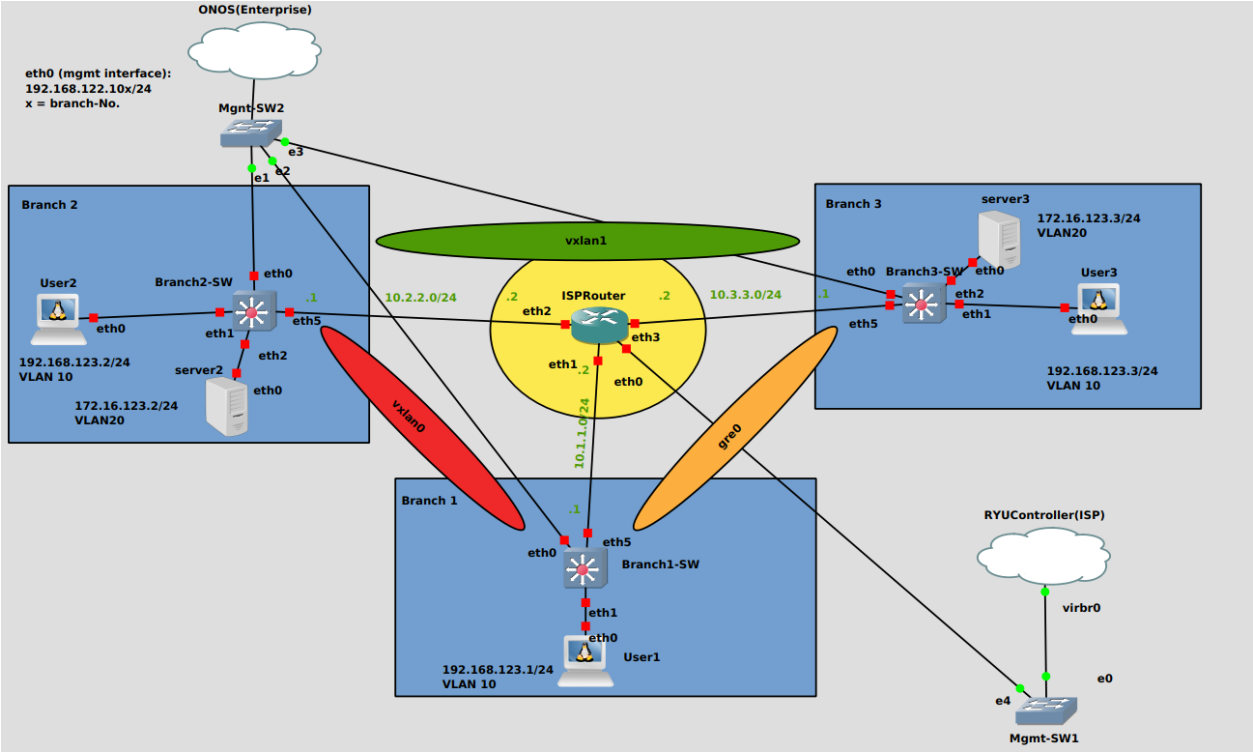

Figure 10. New Servers topology

3) *Testing*

When initiating a ping or Iper3 test, viewing the flow table of a switch through `ovs-ofctl dump-flows br0` reveals flows install in response to the switch forwarding the initial packet to the controller. Once pings or Iperf3 tests complete, the ONOS GUI reveals the new server hosts.

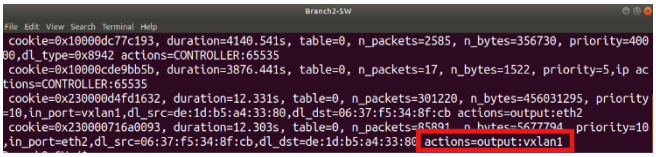

Figure 11. Flow-dump Branch 2 OVS

C. *Branch 1 Network Security*

1) *Design Plan*

A firewall is required to be deployed in Branch 1 protecting the user, this is achieved by manually inserting the required filtering flows. The requirement of the firewall at a high level is to block deny UDP and ICMP packets from `User3`, permit only TCP and ICMP packets from `User2`. Any other `User3` traffic is permitted and any other traffic not permitted from `User2` is denied, egress traffic originating from `Branch 1` is not subject to firewall packet filtering. The firewall is not stateful and therefore replies destined `User1` also will not succeed (e.g. ICMP reply).

2) *Configuration Details*

The appliance used between Branch1-SW is another OVS, however not connected to any controller. All existing bridges are deleted and replaced with new bridge `fw-br`.

- `Ovs-vsctl del-br [bridge]`
- `Ovs-vsctl add-br fw-br`
- `Ovs-vsctl add-port fw-br [interface]`

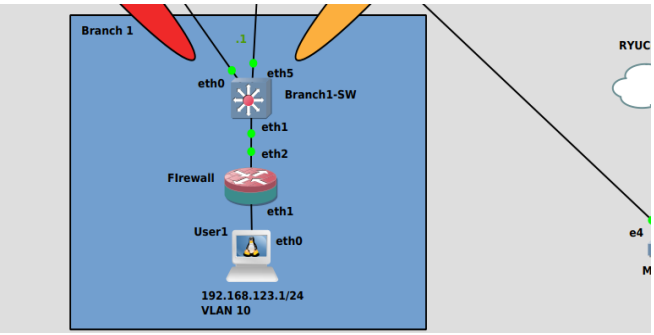

Figure 12. Branch 1 Network Security

Firewall filtering can be implemented by installing flows manually that classify packets based on packet attributes and process them according to an action. The following flows address SHS2 security requirements: (Note: `ovs-ofctl add-flow fw-br table=0` prefixes each command)

(1) *`Priority=3000,udp,in_port=eth2,nw_src=192.168.123.3,action=drop`*

(2) `Priority=3000,icmp,in_port=eth2,nw_src=192.168.123.3,actions=drop`
(3) `Priority=3000,tcp,in_port=eth2,nw_src=192.168.123.2,actions=NORMAL`
(4) `priority=3000,icmp,in_port=eth2,nw_src=192.168.123.2 actions=NORMAL`
(5) `priority=2000,ip,in_port=eth2,nw_src=192.168.123.3 actions=NORMAL`
(6) `priority=2000,ip,in_port=eth2,nw_src=192.168.123.2 actions=drop`

3) *Testing*

Iperf3 and ping tests can verify firewall functionality and ensure expected firewall behaviour is conformant to the requirements. (Note: The flows corresponding index number has no corresponding relation to the behaviour and implications of flows and it used for reference in the documentation of testing)

(1) *Use Iperf3 and conduct test with UDP option set*
Block UDP from branch 3 host
- *User1:* `iperf3 -s`
- *User3:* `iperf3 -c 192.168.123.1 -u`
- ***Expected result:*** `iperf3: error - unable to read from stream socket: Resource temporarily unavailable`

(2) *Use ping command from User3*
Block ICMP from branch 3 host
- *User3:* `ping 192.168.123.1 -c 4`
- *Expected result:* `4 packets transmitted, 0 received, 100% packet loss`

(3) *Use Iperf3 and conduct test with TCP option set*
Allow other traffic from branch 3 host
- *User1:* `iperf3 -s`
- *User3:* `iperf3 -c 192.168.123.1`
- ***Expected result:*** `Successful connection test`

(4) *Use Iperf3 and conduct test with TCP option set*
Allow TCP from branch 2 host
- *User1:* `iperf3 -s`
- *User2:* `iperf3 -c 192.168.123.1`
- ***Expected result:*** `Successful connection test`

(5) *Use ping command from User2*
Allow ICMP from branch 2 host
- *User2:* `Ping 192.168.123.1 -c 4`
- ***Expected result:*** `4 packets transmitted, 4 received, 0% packet loss`

(6) *Use Iperf3 and conduct test with TCP option set*
Block all other traffic from branch 2 host
- *User1:* `iperf3 -s`
- *User2:* `iperf3 -c 192.168.123.1 -u`
- ***Expected result:*** `iperf3: error - unable to read from stream socket: Resource temporarily unavailable`

D. *Video Stream Address Translation*

1) *Design Plan*

The server in Branch 2 is and older server and only uses UDP protocol for streaming video on destination port 300. Port 300 is occupied by another service on Server 3, however port 500 is available. Packet manipulation by flows installed on new OVS switch is introduced into Branch 2 in line with traffic flow before Server 2 traffic hits Branch2-SW. OpenFlow protocol enables packet manipulation where the transport

protocol port number can be translated like Port Address Translation (PAT).

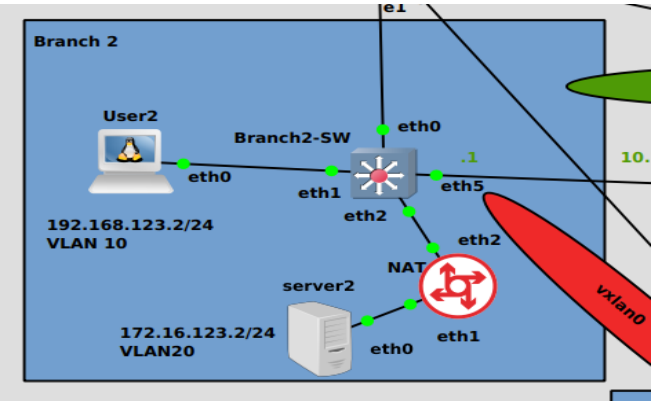

Figure 13. Video Stream Address Translation topology

2) Configuration Details

A dedicated NAT OVS is positioned between Server 2 and Branch2-SW, existing bridges are deleted and a new bridge is configured with both connecting interfaces added to the bridge. Then a flow is installed which captures UDP traffic based on source and destination IP address and modifies the destination port number to value 500 and proceeds with forwarding the packet.

- `Ovs-vsctl del-br [bridge]`
- `Ovs-vsctl add-br nat-br`
- `Ovs-vsctl add-port nat-br [interface]`
- `Ovs-ofctl add-flow nat-br table=0,priority=1000,udp,nw_src=172.16.123.2,nw_dst=172.16.123.3,actions=mod_tp_dst:500,NORMAL`

3) Testing

Socat [8] is used to verify packet manipulation with a small bash script to simulate a constant stream of UDP packets. To create a bash script, use nano to create and edit a shell file (.sh), write into the file the below shell code. Ensure to make the script executable by executing: `chmod +x [file]` (example: `udp_test.sh`).

```
#!/bin/bash
i=1
while true; do
  echo "stream packet $i" | socat - UDP:172.16.123.3:300
  sleep 0.2     # 5 packets per second; adjust for desired rate
  i=$((i + 1))
done
```

Figure 14. UDP connection test script

(1)** Use Socat to set up server 3 listener and server 2 sender **Translate UDP destination port 300 to UDP listening port 500

- *Server2: ./udp_test.sh*
- *Server3:* `socat -v -u UDP4-LISTEN:500,fork STDOUT`
- ***Expected result:*** `stream packet (iteration)`

```
> 2025/10/29 15:06:50.402785  length=16 from=0 to=15
stream packet 3
stream packet 3
> 2025/10/29 15:06:51.114562  length=16 from=0 to=15
stream packet 4
stream packet 4
> 2025/10/29 15:06:51.827567  length=16 from=0 to=15
stream packet 5
stream packet 5
```

Figure 15. Expected result seen on server 3

The script is executing a UDP connection on port 300 and the expected result is the stream packet combined with its iteration count. Server 3 listens on port 500 allowing multiple unidirectional connections. Wire shark captures can further verify port number translation.

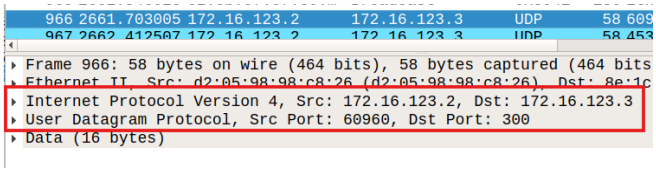

Figure 16. Server2 eth0 to NAT eth1

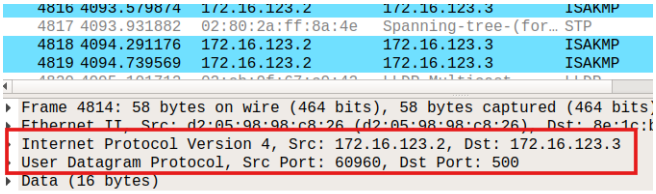

Figure 17. NAT eth2 to Branch2-SW eth2

4) Discussion

Initial planning saw testing to be conducted with Iperf3, however some issues with the establishment of Iperf3 testing made testing problematic packet manipulation. The initial connection of all Iperf3 tests is still dependent on a TCP connection. Test conditions with Server 3 listening on port 500 and then initiating a client connection from server to using port 300 with the UDP option set. With appropriate packet manipulating flows, it failed to establish testing with UDP packets.

E. Network Monitoring

1) Design Plan

NTOPGN [9] is a full-visibility network monitoring tool that can congregate networking logs and provides visualizations and analysis. Traffic passing through `Branch3-SW` is mirrored to NTOPGN to the local VM through the cloud appliance. Port mirroring does not affect the intended connection and is transparent to sending and receiving hosts.

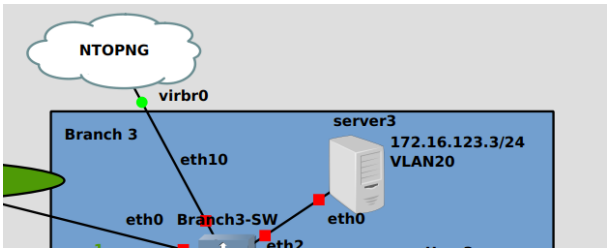

Figure 18. Networking Monitoring Topology

2) *Configuration Details*

NTOPNG is ran locally on the host VM, the cloud appliance is utilised exactly how `mgmt-switches` and OVS connect to the ONOS and Ryu controllers seen in part A. The `virbr0` is configured on the cloud appliance and `eth10` is configured to be part of the bridge; `br0` on `Branch3-SW`, see *Figure 17*. Port mirroring can be configured through CLI command as seen below:

- `Ovs-vsctl add-port br0 eth10`
- `ovs-vsctl -- set bridge br0 mirrors=@m -- --id=@eth10 get port eth10 -- --id=@m create mirror name=NTOPNG-mirror select-all=true output-port=@eth10`

The installation guide for NTOPNG can be found in the documentation [9]. There are no Debian builds that are compatible with the host VM Ubuntu version. Docker containers can are used instead to host NTOPNG. The NTOPNG container depends on Redis which is an in-memory cache database [10]. Create a directory and create a docker-compose file as below in *Figure 19*:

```
version: "3.3"
services:
  ntopng:
    container_name: ntopng
    image: ntop/ntopng:latest
    command: --community -d /var/lib/ntopng -r
127.0.0.1:6379@0 -w 0.0.0.0:3000
    volumes:
      - ./data/ntopng:/var/lib/ntopng
    network_mode: host
    restart: unless-stopped
  redis:
    container_name: ntopng-redis
    image: redis:alpine
    command: --save 900 1
    ports:
      - "6379:6379"
    volumes:
      - ./data/redis:/data
    restart: unless-stopped
```

Figure 19. Sample docker-compose file [10]

- `mkdir ntopng`
- `nano docker-compose.yml`
- `docker-compose up`

Docker-compose [12] may need to be installed in order to execute the compose up command (`sudo apt install docker-compose`). Ensure that the docker-compose file is created within the ntopng directory and start-up command is also executed within that working directory. Enter the IP of the VM on port 3000 in a web browser to find the NTOPNG application running (e.g. 192.168.211.143:3000 or localhost:3000). In the case of my deployment interface `gns3tap0-4` was the interface that should be selected in the GUI to view mirrored traffic. Running a continuous ping to `User3` and examining each interfaces traffic can verify that ICMP are received on this interface.

3) *Testing*

Wireshark capture on link on the connection between the cloud appliance (`Virbr0`) and `Branch3-SW` confirms ICMP packets are mirrored, see *Figure 20* (top of image). Furthermore, it is directly evident in *Figure 21*.

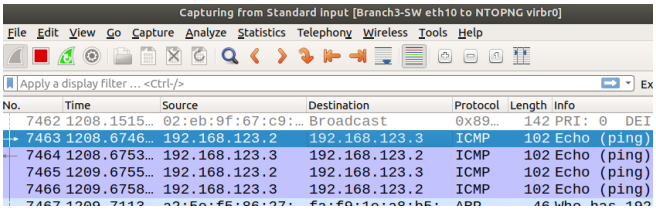

Figure 20. Wireshark capture of port mirroring

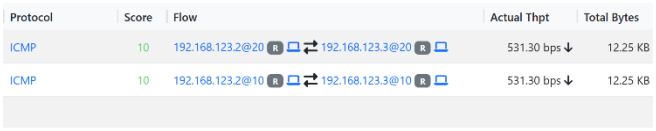

Figure 21. NTOPNG live flows view on gns3tap0-4

Testing with Iperf3 to generate elephant traffic (high bandwidth utilisation) can be completed and viewed in the NTOPNG GUI in the live flows section. Iperf3 execution options `-P` indicates number of parallel streams to run to the server and `-t` indicates time in seconds for the duration of the test. To enable multiple Iperf3 tests to run concurrently, two unique ports need to be allocated and run in daemon mode.

- *User3:* `iperf3 -s -p 5201 -D`

  `iperf3 -s -p 5202 -D`
- *User1:* `iperf3 -c 192.168.123.3 -P 10 -t 300 -p 5201`
- *User2:* `iperf3 -c 192.168.123.3 -P 10 -t 300 -p 5202`

Confirm Iperf3 is running in the background with the command: `ps aux | grep iperf3`. Command: `kill [pid]` will stop the process. To stop the NTOPNG container, interrupt the container by keying together: `'Ctrl + C'`. Alternatively, enter a new terminal tab and enter: `docker compose stop`.

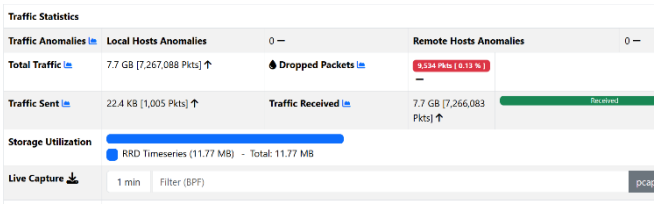

Figure 22. NTOPNG gns3tap0-4 interface details

4) Discussion

NTOPNG was not installed locally because it has no version compatible with Ubuntu 18.04.5, however docker containers were used instead which runs the application with the dependencies isolated from the host system (container still has configuration to communicate on a network level i.e. NTOPGN analysing interfaces and traffic). Iperf3 requires additional execution options to handle concurrent connections on one host. This was discovered by reading the help options of Iperf3 and trial and error testing.

F. Network Automation

1) Design Plan

GNS3 restarts causes appliances configuration to reset, there are some configurations that can be made to persist however some identified reoccurring configurations that are required on restarts are; ONOS and Ryu controller startups, ISP router interface configuration through rest router API, MTU size for ISP router physical interfaces, static NAT and firewall flows. A Python script is made to automate ONOS and Ryu controller start up and ISP router interface configuration through rest router.

2) Configuration Details

The full code contents can be seen in *Appendix 1,* Python 2.7.17 is the version for the following code. The flow of the program is handled by five main functions that automate the minimum requirements. When starting controllers the output is verbose and contains many lines of logs. To prevent this, the processes logs are redirected to Python's special constant `os.devnull` which acts as a black hole for the logs.

```
def start_controllers():
    FNULL = open(os.devnull, 'w')
    onos = subprocess.Popen(
        ["sudo", "/opt/onos/bin/onos service",
"start"],
        stdout=FNULL,
        stderr=subprocess.STDOUT,
    )
```

Figure 23. Start controllers code snippet

The `requests` package [11] is a library that has easily implemented functionality for making various types of HTTP method requests. As seen in prior sections where manually curl commands are used to contact the Ryu rest router API. In the code GET request is used to retrieve the ISP router device ID after connecting to the Ryu controller. A POST request is then made to install the virtual interfaces on the ISP router.

```
r =
requests.get("http://localhost:8080/stats/switch
es")
if r.status_code == 200:
    r = r.json()
    print("%016x" % r[0])
    return ("%016x" % r[0])
```

Figure 24. Get ISP router ID code snippet

The expected return of the request is a list with integers that need to be transformed in 16-digit hex value for the rest router API to be able to process. The return value is passed in as a parameter for the next function in the program which configures the ISP router links with the IP addresses. (Note: localhost translates to IPv4 address 127.0.0.1)

```
ipv4 = ["10.1.1.2/24", "10.2.2.2/24",
"10.3.3.2/24"]
    for i in range(len(ipv4)):
        payload = {"address": ipv4[i]}
        url = "http://localhost:8080/router/%s"
%sw_id
        r = requests.post(url,
data=json.dumps(payload))
```

Figure 25. Configure ISP interface IP addresses

Figure 25 shows the building of the HTTP request that is made to configure the interfaces through a POST method for each IP in the list. The final function the program is to run a check ISP router, the expected return is output displaying the three added interfaces, see *Appendix 1,* `def test_ISP_interfaces(sw_id)`

3) Testing

The program is written in mind for basic testing by examination of the programs output. *Figure 26* shows the expected output for successful completion of the program. Further confirmation of controller start up completion is issuing the CLI command: `sudo lsof -i -P -n | grep LISTEN` and looking for Ryu and ONOS listening ports.

```
Getting switch ID
0000661063781143
('Success:', u'[{"switch_id": "0000661063781143", "command_result": [{"result": "success",
 "details": "Add address [address_id=1]"}]}]')
('Success:', u'[{"switch_id": "0000661063781143", "command_result": [{"result": "success",
 "details": "Add address [address_id=2]"}]}]')
('Success:', u'[{"switch_id": "0000661063781143", "command_result": [{"result": "success",
 "details": "Add address [address_id=3]"}]}]')

running: curl http://localhost:8080/router/<switch-id>
('\nVerifying ISP interface config: ', u'[{"switch_id": "0000661063781143", "internal_netw
ork": [{"address": [{"address_id": 1, "address": "10.1.1.2/24"}, {"address_id": 2, "addres
s": "10.2.2.2/24"}, {"address_id": 3, "address": "10.3.3.2/24"}]}]}]')
ONOS API up
{"time":8167121376246,"devices":3,"links":0,"clusters":3}
```

Figure 26. Expected script output

```
url = "http://localhost:8181/onos/v1/topology"
    auth = HTTPBasicAuth("onos", "rocks")
    while retries > 0:
        try:
            r = requests.get(url,auth=auth,
timeout=30)
```

Figure 27. Test ONOS controller

Figure 27 shows snippet of code that makes a GET request to ONOS API and returns the topology devices connected to the ONOS controller. Basic authentication is required when accessing ONOS API endpoints. A similar request is made for

the Ryu controller which queries for the added interfaces previously done in the script, however it does not require any authentication. The sleep timers allow both controllers to become stable before executing any requests. See *Figure 26* in the bottom half of the output for expected results from the requests.

4) Discussion

The program does not terminate the Ryu controller process when interrupting the program. Attempts were made however there were issues regarding inconsistent ownership of processes. ONOS is executed under sudo however Ryu startup is executed by the sdn user. This results in processing handling and signalling issues in the terminate of the program. Attempts were to resolve the issue, however not without encountering other issues as a result. Therefore, when stopping the program, the user is required to check and manually kill the Ryu and ONOS processes. Use command: `ps aux | grep [onos | Ryu]` to identify process ID. In the beginning of writing the script, it was planned to be more dynamic, in that it could be more applicable to similar topologies. However, realization that identifying which switch needs specific IP address would require either extensive execution parameters or user input. This order of thinking was not continued, and in favour a less complex automation script was written. This is why some aspects of the code may perform a loop for no real reason given it may iterate once.

Some issues that have been encountered with controller startup. Verifications checks can determine that the controller is up but not exposing listening ports or does not at start correctly. This occurs most often with the ONOS controller, and the simplest solution is to delete ONOS and create a fresh deployment. Following this, you must again change ONOS listening port number to 7777.

- `$ cd /opt`
- `$ sudo rm-rf onos`
- `$ sudo tar xzf onos-2.2.3.tar.gz`
- `$ sudo mv onos-2.2.3 onos`

IV. REFERENCES

[1] Cloudflare, *"What is the control plane?"* Cloudflare Learning Center. [Online]. Available: https://www.cloudflare.com/learning/network-layer/what-is-the-control-plane/.

[2] GNS3 Technologies, *"Getting Started with GNS3,"* GNS3 Documentation. [Online]. Available: https://docs.gns3.com/.

[3] Ryu Project, *"Welcome to Ryu the Network Operating System (NOS),"* Ryu 4.34 Documentation. [Online]. Available: https://ryu.readthedocs.io/en/latest/.

[4] ONOS Project, *"ONOS – Open Network Operating System,"* ONOS Wiki. [Online]. Available: https://wiki.onosproject.org/.

[5] iPerf Project, *"iPerf3 and iPerf2 user documentation,"* ESnet. [Online]. Available: https://iperf.fr/.

[6] Arista Networks, *"VXLAN Use Cases in Cloud Scale Data Centers,"* Technical White Paper. [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/VXLAN_Use_Cases.pdf.

[7] Cloudflare, *"What is GRE tunneling? How GRE protocol works,"* Cloudflare Learning Center. [Online]. Available: https://www.cloudflare.com/learning/network-layer/what-is-gre-tunneling/.

[8] Socat Project, *"socat(1) – Multipurpose relay (SOcket CAT),"* Linux Manual Pages. [Online]. Available: https://linux.die.net/man/1/socat.

[9] ntop, *"What is ntopng,"* ntopng 6.5 Documentation. [Online]. Available: https://www.ntop.org/guides/ntopng/.

[10] J. Lalib, *"docker-ntopng/docker-compose.yml,"* GitHub Repository. [Online]. Available: https://github.com/JLalib/docker-ntopng/blob/main/docker-compose.yml.

[11] Kenneth Reitz, *Requests: HTTP for Humans™ —* Documentation. [Online]. Available: https://requests.readthedocs.io/en/latest/.

[12] Docker Inc., *Docker Compose — Docker Docs*. [Online]. Available: https://docs.docker.com/compose/

V. Appendix

```
import requests, time, json
import os, sys, signal, subprocess
from requests.auth import HTTPBasicAuth

def main():
    start_controllers()
    sw_id = get_switch_id()
    add_ISP_interfaces(sw_id)
    test_ISP_interfaces(sw_id)
    test_onos()

def start_controllers():
    FNULL = open(os.devnull, 'w')
```

```
    onos = subprocess.Popen(
        ["sudo", "/opt/onos/bin/onos-service", "start"],
        stdout=FNULL,
        stderr=subprocess.STDOUT,
    )
    Ryu = subprocess.Popen(
        ["Ryu-manager", "--verbose", "rest_router.py", "gui_topology/gui_topology.py"],
        cwd="/home/sdn/Downloads/Ryu/Ryu/app/",
        stdout=FNULL,
        stderr=subprocess.STDOUT,
    )

def get_switch_id():
    while True:
        time.sleep(10)
        try:
            print("\nGetting switch ID")
            r = requests.get("http://localhost:8080/stats/switches")
            if r.status_code == 200:
                r = r.json()
                print("%016x" % r[0])
                return ("%016x" % r[0])
        except Exception as e:
            print(e)

def add_ISP_interfaces(sw_id):
    ipv4 = ["10.1.1.2/24", "10.2.2.2/24", "10.3.3.2/24"]
    for i in range(len(ipv4)):
        payload = {"address": ipv4[i]}
        url = "http://localhost:8080/router/%s" %sw_id
        r = requests.post(url, data=json.dumps(payload))
        if r.status_code == 200:
            print("Success:", r.text)
        else:
            print("Failed:", r.status_code, r.text)

def test_ISP_interfaces(sw_id):
    url = "http://localhost:8080/router/%s" %sw_id
    r = requests.get(url)
    if r.status_code == 200:
        print("\nrunning: curl http://localhost:8080/router/<switch-id>")
        print("\nVerifying ISP interface config: ", r.text)

def test_onos():
    retries = 3
    url = "http://localhost:8181/onos/v1/topology"
    auth = HTTPBasicAuth("onos", "rocks")
    while retries > 0:
        try:
            r = requests.get(url, auth=auth,timeout=30)
            if r.status_code == 200:
                print("ONOS API up")
                print(r.text)
                break
            else:
                print("waiting for ONOS...")
                retries -= 1
        except Exception as e:
            print(e)
            time.sleep(10)

if __name__ == "__main__":
    main()
```

Appendix 1. Base automation Python code